GUI Behavior from Source Code Analysis

João C. Silva[1,2] Carlos E. Silva[1] José C. Campos[1]
João A. Saraiva[1]
[1] Departamento de Informática/CCTC, Universidade do Minho
[2] Departamento de Tecnologia, Instituto Politécnico do Cávado e do Ave
{jose.campos,jas}@di.uminho.pt, jcsilva@ipca.pt, carlosebms@gmail.com

Abstract
When developing interactive applications, considering the correctness of graphical user interfaces (GUIs) code is essential. GUIs are critical components of today's software, and contemporary software tools do not provide enough support for ensuring GUIs' code quality. GUIsurfer, a GUI reverse engineering tool, enables evaluation of behavioral properties of user interfaces. It performs static analysis of GUI code, generating state machines that can help in the evaluation of interactive applications. This paper describes the design, software architecture, and the use of GUIsurfer through an example. The tool is easily re-targetable, and support is available to Java/Swing, and WxHaskell. The paper sets the ground for a generalization effort to consider rich internet applications. It explores the GWT web applications' user interface programming toolkit.

Keywords
Graphical user interface, Reverse Engineering, Analysis

1 Introduction

Practice shows that the user interface layer of interactive applications is the one more likely to suffer changes during the life-time of an application. Available technology to build user interfaces mostly consists of libraries of components that are glued together in an event-based style of programming, leading to code that is hard to understand and maintain. For example, the fact that Swing [Walrath 04] components are based on the Model-View-Controller (MVC) architectural pattern, does not necessarily mean that the pattern is maintained at application level. Indeed, the source code of Swing-based user interfaces can quickly become a collection of method calls accessing different parts of some common global state, reviving the notion of "spaghetti code". The multitude of technologies and frameworks being made available for developing Web applications, if anything, is making things worst [Mikkonen 07].

Integrated Development Environments (IDEs) help developers in creating the user interfaces by allowing them to *draw* the user interface, and attach methods to relevant object/events. However, they do not necessarily promote better code structuring and quality.

The above two issues, the need to constantly change code, and poor code quality from the start, mean that a considerable effort needs to be invested into the development and maintenance of the GUI layer of applications. Tools are needed that can help in this process. Reverse engineering tools, in particular, can have a role in helping analyse, understand, and manipulate source code.

Our objective consists in developing tools to automatically extract models from GUI source code. The extracted models should specify, for example, when a particular GUI event can occur, which are the related conditions, which system actions are executed or which GUI state is generated next. We want to be able to reason about, and test, this GUI model in order to analyze aspects of the original application's usability, and the quality of the implementation. Additionally, we want the developed tools to be language independent. Through the use of generic techniques, the tool enable to analyze different source code paradigms, such object oriented or functional. This work will not only be useful to enable the analysis of existing interactive applications, but can also be helpful when an existing application must be ported or simply updated [Melody 96].

In previous papers [Silva 06, Silva 09] we have explored the applicability of slicing techniques to our reverse engineering needs, and developed the building blocks for the approach. In this paper we describe our tool making use of two Agenda interactive application as running examples. A new module is presented, allowing us to analyze GWT-based rich internet applications.

The paper is organised as follows: section 2 describes the running example; section 3 describes the reverse engineering approach; section 4 describes some of the models the tool is able to generate; section 5 discusses analysis; section 6 discusses rich internet applications' support; section 8 discusses related work; and section 9 concludes with

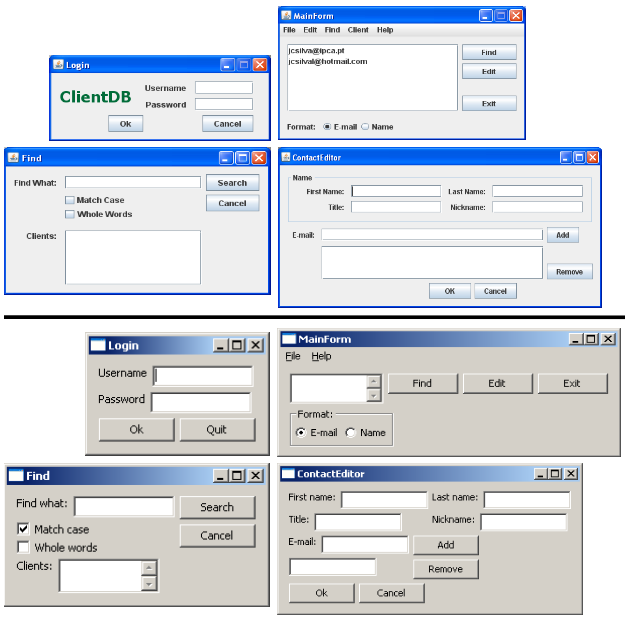
jcsilva@ipca.pt
jcsilval@hotmail.com

Figure 1. Two agenda applications — Java/Swing (top) and WxHaskell (bottom)

some pointers to future work.

2 The Agenda Example

Throughout the paper we will use two Agenda interactive applications as running examples. The first one was implemented with Java/Swing [Loy 02]. The second in WxHaskell [Jones 99]. These applications implement an agenda of contacts: they allows users to perform the usual actions of adding, removing and editing contacts. Furthermore, they also allows users to find a contact through its name.

Each agenda consists of four windows, named *Login*, *MainForm*, *Find* and *ContactEditor*, as shown in Figure 1. The initial *Login* window (Figure 1, top-left window) is used to control users' access to the agenda. Thus, a login and password have to be introduced by the user. If the user introduces a valid login/password and presses the `Ok` button, then the login window closes and the main window of the application is displayed. On the contrary, if the user introduces an invalid login/password, then the input fields are cleared, a warning message is produced and the login window continues to be displayed. By pressing the *Cancel* button in the *Login* window, the user exits the application.

Authorized users can use the main window (Figure 1, top-right window) to find and edit contacts (*Find* and *Edit* buttons). By pressing the *Find* button in the main window, the user opens the *Find* window (Figure 1, bottom-left window). This window is used to search and obtain a particular contact's data from his name. By pressing the *Edit* button in the main window, the user opens the *ContactEditor* window (Figure 1, bottom-right window). This last window allows the edition of all contact data, such as name, nickname, e-mails, etc. The *Add* and *Remove* buttons enable edition of the e-mail addresses list of the contact. If there are no e-mails in the list then the *Remove* button is automatically disabled.

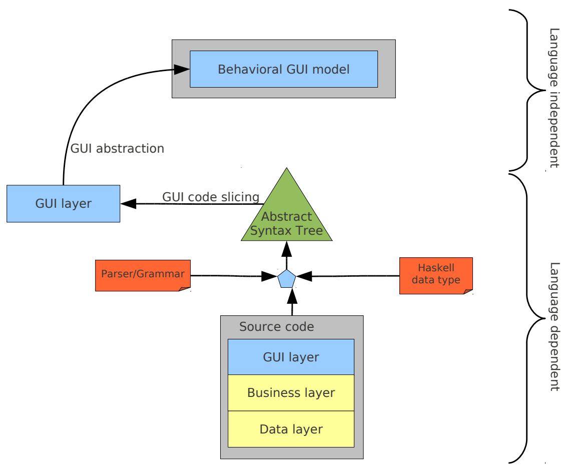

Figure 2. Model-based GUI reasoning process

3 GUI Models from Source Code

In order to achieve our goal of developing an approach for reverse engineering of GUI source code, we resorted to several techniques. Figure 2 describes our approach.

Using a parser, an Abstract Syntax Tree (AST) is obtained from the source code. Then we identify all fragments in the abstract syntax tree that are members of the GUI layer. This is achieved through program slicing [Tip 95]. We use the GUI constructors to focus the slicing in the subtrees that represent the GUI. Slicing is based on the program dependency graph.

The *GUI code slicing* module extracts graphical user interface AST fragments through code slicing. This is a generic module to extract GUI fragments from any AST, i.e. Java/Swing, wxHaskell, C#, etc. This allows us to identify all of the program fragments that interact with the graphical user interface. We do a traversal of the tree (based on the program dependency graph), and detect all GUI nodes.

In order to extract the user interface behavior from the source code of the interactive applications, we need to construct a slicing function that isolates a sub-program from the entire program. Because we want to reuse our approach across different programming languages and paradigms, we need to use generic techniques that work for *any* AST and not for a particular language only.

Using strategic programming [Visser 03, Visser 04] we make use of a pre-defined set of (strategic) generic traversal functions that traverse any AST using different traversal strategies (e.g. top-down,left-to-right, etc). Thus, the programmer has to focus in the nodes of interest, only. In fact, the programmer does not need to have a knowledge of the entire grammars/AST, but only of those parts he is interested in (e.g., the Swing sub-language). As a result, he/she does not need full knowledge of the grammar to write recursive functions that isolate the graphical user interface sub-program from the entire program. We used *Haskell* to develop a GUI code slicing library which contains a generic set of traversal functions that traverse any AST.

Figure 2 shows also that the *GUI abstraction* module uses the GUI fragments to produce a behavioral user interface description (the *Behavioral GUI model*). The fragments relevant to the GUI reverse engineering are limited to graphical user interface instructions, control flow information, and methods invocation. From these fragments of the original AST it is finally possible to extract the GUI layer and reason about it.

The implemented prototype is language independent in what regards the strategic programing and slicing techniques. The prototype is language dependent with respect to the grammar and program dependency graph definition.

4 GUI Models

The abstractions that we look for in the source code are widgets that enable users to input data (*user input*), widgets that enable users to choose between several different options such as a command menu (*user selection*), any actions that are performed as the result of user input or user selection (*user action*), and any widget that enables communication from application to users such as a user dialogue (*output to user*). Given the user interface code of an interactive system and this set of abstractions, we can generate its graphical user interface abstraction.

Currently, the tool is capable of automatically generating models of the interface. Next we describe examples that we can automatically generate from the Agenda applications' source code.

The first example, presented in Figure 3, is a directed graph describing the Agenda application behavior. Interactive systems can be represented as directed graphs [Melody 96]. User actions are mapped into arcs and states are GUI application idle time. When the user performs an action, the current state A is changed to the next state B where there is a directed arc from A to B labeled with that action. Arcs may point back to the same state, and the transition then does not change the state. Graph models may be non-deterministic because of the underlying system, in which case one of several possible next states will be arrived at.

The directed graph presented in Figure 3 is useful to visualize application's windows states. Each state of this model contains the description of all windows opened in a particular period of time. Transitions between states correspond to events that allow to open or close windows. Each transition refers first the window's name, it state status, the event and respective condition. In this case, we can reason about which windows can be opened along a session, which are the related events, and which conditions must hold. As an example, at the top left corner, this model specify the Login window as the entry point for the application. Then, from the Login window, there is one transition to the MainForm window through the Ok event and cond2 condition pair (each event and condition identifiers are related to the respective source code of the original application). The referred transition is:

$$Login\ state1\ Edit\ cond2$$

The state1 identifier refers to the internal state of the Login window. Although not illustrated in this paper, it is possible to generate models representing how the internal state of a windows changes. From MainForm, it is possible, for example, to open the ContactEditor window through the Edit/cond2 pair.

In this particular case the Java and Haskell Agenda applications have the same behavior. Hence, the graph from Figure 3 can be obtained from both. As an example, let us consider the following Java/Swing and WxHaskell source code which opens a new ContactEditor window:

```
...
private void EditActionPerformed(...) {
  new ContactEditor().setVisible(true);
  ...
```

and

```
...
edit <- button pn [
text := "Edit",
on command := start contactEditor
]
...
```

This source code is abstracted to the states in Figure 3 that make reference to the ContactEditor form.

Another example of a model generated by GUIsurfer is presented in Figure 4. This model contains all possible states for each application window. The model presents also the total number of events associated to each state transition. This is useful as a metric to detect windows complexity.

These examples provide an indication of how the developed prototype achieves our main objective. It generates, automatically, a GUI behavioral model directly from an application's source code. The behavioral model describes window states, events, conditions and actions.

We now want to be able to reason about this GUI model. The techniques described in this paper enable us to analyse properties of the interface. For example, we can use graph-based algorithms to compute if all states are accessible from the initial one, in order to detect whether a particular window of the application will ever be displayed or not. We can also produce valid or invalid *sentences* of the language defined by the machine to use as test cases. These test cases can be used to prove more advanced properties of the interface. The next section will show how we can reason about the original application's characteristics.

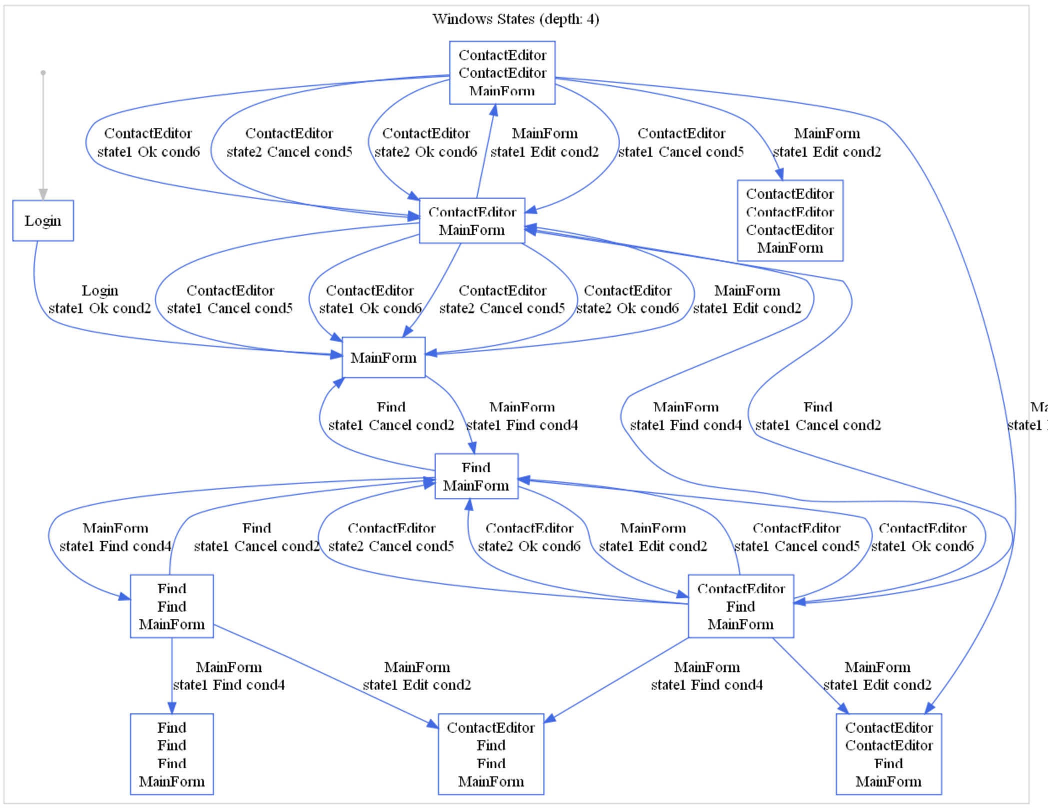
Windows States (depth: 4)
ContactEditor ContactEditor MainForm
ContactEditor state1 Ok cond6
ContactEditor state2 Cancel cond5
ContactEditor state2 Ok cond6
MainForm state1 Edit cond2
ContactEditor state1 Cancel cond5
MainForm state1 Edit cond2
Login
ContactEditor MainForm
ContactEditor ContactEditor ContactEditor MainForm
Login state1 Ok cond2
ContactEditor state1 Cancel cond5
ContactEditor state1 Ok cond6
ContactEditor state2 Cancel cond5
ContactEditor state2 Ok cond6
MainForm state1 Edit cond2
MainForm
Find state1 Cancel cond2
MainForm state1 Find cond4
MainForm state1 Find cond4
Find state1 Cancel cond2
MainForm state1 Find cond4
Find MainForm
MainForm state1 Find cond4
Find state1 Cancel cond2
ContactEditor state2 Cancel cond5
ContactEditor state2 Ok cond6
MainForm state1 Edit cond2
ContactEditor state1 Cancel cond5
ContactEditor state1 Ok cond6
Find Find MainForm
ContactEditor Find MainForm
MainForm state1 Find cond4
MainForm state1 Edit cond2
MainForm state1 Find cond4
MainForm state1 Edit cond2
Find Find Find MainForm
ContactEditor Find Find MainForm
ContactEditor ContactEditor Find MainForm

Figure 3. Agenda GUI state machine

5 Analysis

The reverse engineering approach described in this paper allows us to extract an abstract GUI behavior specification. Now, GUI analysis can be performed.

Graphs are a mathematical concept that can be used to model Graphical User Interfaces [Thimbleby 08]. Sequences of user actions are paths in a graph. A standard graph theoretic concept is the shortest path between two edges, which defines the most efficient way a user can achieve a particular change of state. Graphs define interactive systems and usability properties.

Using graph models we extended our prototype implementing graph operators. At this time we have implemented intersection, union and difference of graphs. This is particularly useful to compare versions of an application, allowing to determine different versions have the same behavior.

Let us consider a new version of the Java/Swing Agenda application without the Contact Editor form. I.e., without the following Java/Swing instruction.

```
new ContactEditor().setVisible(true);
```

Using the difference graph operator, we are able to obtain behavioral differences between applications. For example, calculating the difference between the WxHaskell and Java/Swing versions of the Agenda, we obtain the graph in Figure 5. The obtained graph identifies the actions which can only be executed in the WxHaskell Agenda implementation.

Additionally, we can make use of the *QuickCheck Haskell* library tool. QuickCheck [Claessen 00] is a tool for testing *Haskell* programs automatically. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators. GUIsurfer is capable of generating a *Haskell* program capturing the behaviour of the application. Then properties may be defined and QuickCheck may be used to test them in a large number of randomly generated cases.

6 Rich Internet Applications

RIAs are an emergent technology whose primary goal is to develop web applications with the strengths of desktop applications. The principal advantages of desktop applications in comparison to traditional web applications are [Silva 10]:

- absence of page reloading;
- no need for an on-line connection;
- easy interaction with other desktop applications;
- superior interaction experience and usability;

However, traditional web applications, applications accessed through the network(internet/intranet), also have specific advantages such as:

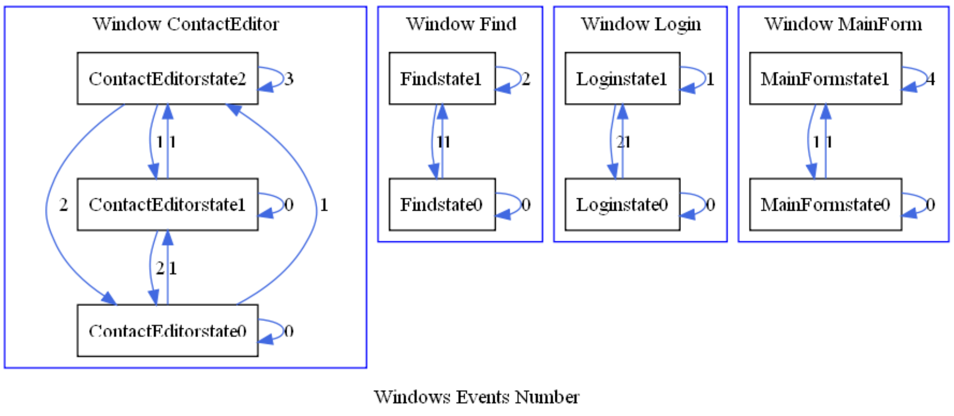

Figure 4. GUI state events number

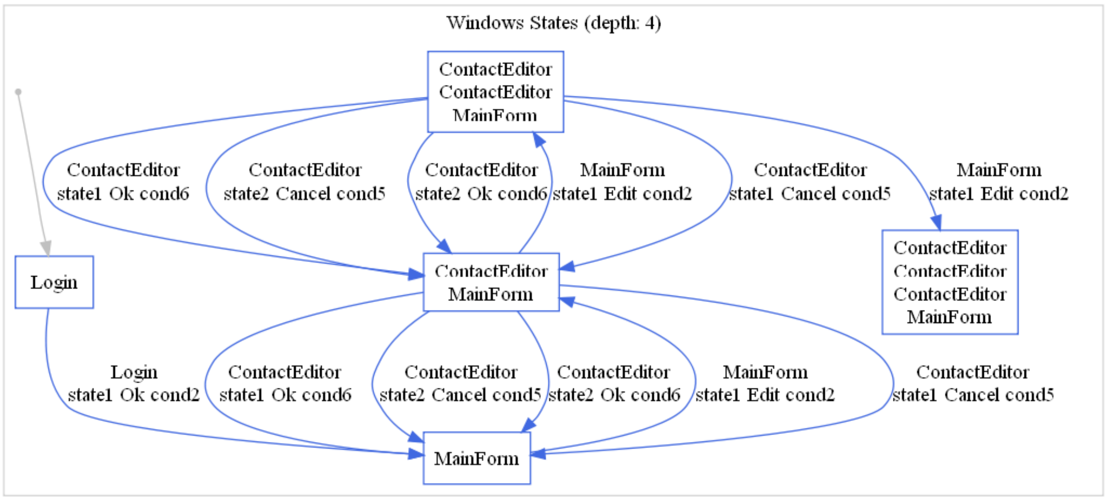
Windows States (depth: 4)
ContactEditor ContactEditor MainForm
ContactEditor state1 Ok cond6
ContactEditor state2 Cancel cond5
ContactEditor state2 Ok cond6
MainForm state1 Edit cond2
ContactEditor state1 Cancel cond5
MainForm state1 Edit cond2
Login
ContactEditor MainForm
ContactEditor ContactEditor ContactEditor MainForm
Login state1 Ok cond2
ContactEditor state1 Ok cond6
ContactEditor state2 Cancel cond5
ContactEditor state2 Ok cond6
MainForm state1 Edit cond2
ContactEditor state1 Cancel cond5
MainForm

Figure 5. Comparing version of an application (using difference operator)

- they do not require deployment/installation or updates in every desktop;
- they provide easier access since one just needs an internet connection;
- they are easily made available in more platforms;
- concentrated information eases security and backups processes;

RIA technologies attempt to bring together these two worlds.

The question, then, is whether GUIsurfer can be extended to deal with this new and emerging programming paradigm.

6.1 Google Web Toolkit

Google Web Toolkit (GWT) is a technology that allows for the development of rich internet applications using the Java programming language. Since GUIsurfer is already able to reverse engineer Java/Swing programs, extending it to deal with GWT seemed like a logical next step.

As it can be deduced by its name, GWT was developed and introduced by Google (version 1.0 was released in May 2006). GWT is a set of development tools, programming utilities, and widgets that enable the user to create rich internet applications. Developing the application in the Java language allows GWT to bring all of Java's benefits to RIAs. Consequently, GWT aims to make the coding of RIAs as simple as possible while allowing interaction with existing JavaScript code.

GWT's goal is to make it easy to develop complex cross-browser applications. To this end, GWT provide a set of ready-to-use user interface widgets that can be immediately utilize to create new applications. Moreover, it also provides a simple way to create original widgets by combining the existing ones. Since GWT produced a JavaScript application, it does not require browser plug-ins additions, and there is also no need for an application server if the applications runn fully on the client-side.

By making the development in the Java programming language, GWT inherits many of Java benefits. One of these benefits is that it provides better application management (when compared to plain JavaScript), thus making GWT a proper solution for the development of Web applications with significant size. This occurs as Java is an object oriented language, therefore allowing Java projects to generally be easy to communicate and comprehend.

Another advantage of using the Java language arises as it enables using any Java Integrated Development Environment (IDE) during application development. A Java IDE improves development as they provide several tools to help developers, for instance, code completion or error checking, and even tools to help debugging the application. Moreover, by using the Java language, one benefits also from Java type checking, therefore decreasing the number of application errors. There is also an improvement on JavaScript debugging, as errors are noticed in compilation time instead of execution time.

6.2 GWT support in GUIsurfer

RIAs have a far greater division between the interface code and the rest of the code, since they divide the code into client-side and server-side. This division is important since it helps GUIsurfer perform GUI code slicing, as the code is more partitioned. GUIsurfer possesses an architecture with a well defined distinction between the language dependent modules and language independent modules. The goal of generalizing it to GWT is made easier because there's only the need of restructuring the language dependent modules.

Since GWT it is a Java toolkit, the same parser already used by GUIsurfer for Java/Swing code could be used. Ideally then there would only be the need to perform the slicing step with a different set of GUI components (those of GWT instead of those from Swing). However a few issues arose. The first related to the genericity of the tool and was due to GUIsurfer's original implementation using the $addActionListener$ method of Swing components to identify actions. In GWT however methods are registered though the $addClickHandler$ method. Solving this problem meant parameterizing GUIsurfer on the method used to register event handler in the interface.

A second issue related to differences in the functionality of both toolkits (Swing and GWT). Since a GWT application is a web application, the closing window (in GWT, panels) actions available in Java Swing are not present. Closing a web application is an unusual action, and thus there is no direct support in GWT for doing it, though it can be achieved by invoking native JavaScript. Another issue occurred in detecting a change from a window/panel to another. In Swing this is achieved by invoking the $dispose$ method on a class. In GWT this is achieved by making the visibility attribute of the panels. Again, changes were introduced to address this situation.

Once the above issues were addressed we were able to generate similar models to those on figure 1 for a GWT version of the Agenda application. In this first version of GWT support, an assumption is also made that the GWT code is structured as similar as possible to Java Swing code. Work is currently ongoing and our goal is to loosen these restrictions as much as possible, and generally improve support for panel handling.

7 A Language Independent Tool

In this section the applicability of GUISurfer to GWT and WxHaskell code is discussed. Our retargets to Wx/Haskell and GWT highlight successes and problems with our initial approach. The size of the adaptations, and the time it took to code them are distinct.

The adaptation to GWT was easier because it exploited the same Java parser. The adaptation to Wx/Haskell was more complex as not only the language was different and the same parser could not be used, but also the programming paradigm was different, i.e. Haskell is a functional language. The functional paradigm is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state. For the applicability of GUISurfer to Wx/Haskell we implemented the slicing step for functional programming analysis extracting events, related conditions and GUI actions through WxHaskell syntax. This task is more complex since Wx/Haskell toolkit has a different structure to define GUI components like windows, event actions, etc.

Regarding the applicability of GUISurfer to GWT, we performed the slicing step with the set of GUI components from GWT, which are different those from Swing. Additional structures are different and need slicing adaptation such the addActionListener method of Swing components to identify actions. In GWT the respective method is registered through the addClickHandler method. Changes performed to extend GUISurfer to a new programming language, specifically GWT or WxHaskell, didn't reflect on architectural alterations. Hence, GUISurfer's objective of being a re-targetable tool was accomplished.

8 Related Work

We have described an approach to the analysis of interactive systems from source code. Reverse engineering techniques are used to derive models from the source code of the user interface layer. State machines are used to capture the behaviour of the interface. Graph theory is then used to analyse those state machines.

Having described the approach, we now set it in the general context of current approaches to the reverse engineering, modelling and analysis of interactive systems.

8.1 Reverse engineering

A typical reverse engineering approach is to run the interactive system and automatically record its state and events. Memon et al. [Memon 03] describe an tool which automatically transverses a user interface in order to extract information about its widgets, properties and values. Chen et al. [Chen 01] propose a specification-based technique to test user interfaces. Users graphically manipulate test specifications represented by finite state machines which are obtained from running the system. Systa studies and analyses the run-time behaviour of Java software trough a reverse engineering process [Systa 01]. Running the target software under a debugger allows for the generation of state diagrams.

Another alternative is the use of statical analysis. The reengineering process is based on analysis of the application's code, instead of its execution, as in previous approaches. One such approach is the work by d'Ausbourg et al. [d'Ausbourg 96] in reverse engineering UIL code (User Interface Language – a language to describe user interfaces for the X11 Windowing System, see [Heller 94]). In this case models are created at the level of the events that can happen in the components of the user interface. For example, pressing or releasing a button.

Moore [Moore 96] describes a technique to partially automate reverse engineering character based user interfaces of legacy applications. The result of this process is a model for user interface understanding and migration. The work shows that a language-independent set of rules can be used to detect interactive components from legacy code. Merlo [Merlo 95] proposes a similar approach. In both cases static analysis is used.

We are using static analysis as in [Merlo 95, Moore 96, d'Ausbourg 96]. However, we are applying it to the source code of graphical user interfaces developed in general purpose programming languages, and working on making the approach as language independent as possible.

8.2 Modelling and analysis

State machines and graph theory are common in the modelling and analysis of interactive systems. Horrocks presents a proven technique for designing event-driven software using the statechart notation [Horrocks 99]. With statecharts it is possible to model multiple cross-functional state diagrams within the statechart. Each of these cross-functional state machines can transition internally without affecting the other state machines in the statechart.

Thimbleby [Thimbleby 08] gives examples of the use of graph theory in the modelling and analysis of a real interactive device. The work described a variety of graph theoretic properties, and discuss their significance to interaction design. Graph theory was also proposed for use in human computer interaction in [Memon 01] as a means of analysis. A representation of a GUI component, called an event-flow graph, identifies the interaction of events within a component.

Other work includes using graph theory for providing test models [Lu 08, Li 07]. Automated graphic user interface test models, which are based on the event-flow graph, are proposed.

Our work builds on these approaches to define appropriate models and analysis approaches to the be supported by the GUIsurfer tool.

9 Conclusions

When developing interactive applications, considering the correctness of the graphical user interface code is essential. GUIs are critical components of today's software and contemporary software tools do not provide enough support for guaranteeing GUI code quality and maintenance. With this in mind, we are developing GUIsurfer, a tool to reverse engineer the GUI layer of interactive applications.

This work is an approach for improving analysis techniques allowing us to reason about GUI models through graph theory. We described GUI models extracted automatically from source code, and presented a methodology to reason about the user interface model.

The approach is language-independent. We have applied the techniques to extract similar models from *Haskell/WxHaskell*, *Java/Swing* and *GWT* interactive applications. Theses models enable us to reason about both metrics of the design, and the quality of the implementation of that design. It is not possible with the actual prototype to analyze every existing JAVA/Haskell code. The system assumes source code will be structured according to specific conventions. In this case, it is assumed the code is generated with the conventions used by the NetBeans or WxHaskell integrated development environments (IDEs). To consider other kind of source code structuring, some adjustments would need to be made. Given that IDEs such as Netbeans are widely used, and automatically generate most of the user interface code, we do not believe this to be a major restriction.

Our objective has been to investigate the feasibility of the approach. We now plan to expand the tool at two levels. On the one hand, we will work on improving the GUIsurfer's support to the above mentioned programming languages, and exted it to new languages (for example, directly supporting the analysis of JavaScript code).

On the other hand, we plan to work on incrementing GUIsurfer's models generation capabilities to allow new types of analysis. Currently it produces state models, a type of dialog models. We want to investigate the feasibility of generating task models because they would allow for a more user centered evaluation of the system's design.

Acknowledgements

GUIsurfer development is being carried out in the context of the CROSS and SSaaPPP projects, supported by the Portuguese Research Foundation (FCT) under contracts: PTDC/EIA-CCO/108995/2008 and PTDC/EIA-CCO/108613/2008.

João C. Silva is supported by PhD grant SFRH/BD/30729-/2006 from FCT.

References

[Chen 01] J. Chen e S. Subramaniam. A gui environment for testing gui-based applications in java. *Proceedings of the 34th Hawaii International Conferences on System Sciences*, 2001.

[Claessen 00] Koen Claessen e John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. Em *ICFP, ACM SIGPLAN, 2000*, 2000.

[d'Ausbourg 96] Bruno d'Ausbourg, Guy Durrieu, e Pierre Roché. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. Em *DSV-IS 96*. 1996.

[Heller 94] Dan Heller e Paula M. Ferguson. *Motif Programming Manual*, volume 6A de *X Window System Seris*. O'Reilly & Associates, Inc., second edição, 1994.

[Horrocks 99] Ian Horrocks. *Constructing the User Interface with Statecharts*. Addison-Wesley, Harlow, England, 1999.

[Jones 99] Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the programming language haskell 98. Relatório técnico, Yale University, Fevereiro 1999.

[Li 07] Ping Li, Toan Huynh, Marek Reformat, e James Miller. A practical approach to testing gui systems. *Empirical Softw. Engg.*, 12(4):331–357, 2007.

[Loy 02] Marc Loy, Robert Eckstein, Dave Wood, James Elliott, e Brian Cole. *Java Swing, 2nd Edition*. O Reilly, 2002.

[Lu 08] Yongzhong Lu, Danping Yan, Songlin Nie, e Chun Wang. Development of an improved gui automation test system based on event-flow graph. Em *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, páginas 712–715, Washington, DC, USA, 2008. IEEE Computer Society.

[Melody 96] Moore Melody. A survey of representations for recovering user interface specifications for reengineering. Relatório técnico, Institute of Technology, Atlanta, 1996.

[Memon 01] Atif M. Memon, Mary Lou Soffa, e Martha E. Pollack. Coverage criteria for gui testing. Em *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, páginas 256–267, New York, NY, USA, 2001. ACM.

[Memon 03] Atif Memon, Ishan Banerjee, e Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. Relatório técnico, University of Maryland,USA, 2003.

[Merlo 95] E. Merlo, P. Y. Gagne, J.F. Girard, K. Kontogiannis, L.J. Hendren, P. Panangaden, e R. De Mori. Reengineering user interfaces. *IEEE Software, 12(1), 64-73*, 1995.

[Mikkonen 07] Tommi Mikkonen e Antero Taivalsaari. Web applications - spaghetti code for the 21st century. Relatório Técnico SMLI TR-2007-166, Sun Labs, June 2007.

[Moore 96] M. M. Moore. Rule-based detection for reverse engineering user interfaces. *Proceedings of the Third Working Conference on Reverse Engineering, pages 42-8, Monterey, CA*, november 1996.

[Silva 06] J.C. Silva, José Creissac Campos, e Jo ao Saraiva. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. Em *DSV-IS 2006, Dublin, Irland*. Springer, 2006.

[Silva 09] J.C. Silva, José Creissac Campos, e Jo ao Saraiva. A generic library for gui reasoning and testing. *SAC ACM, Honolulu, USA*, March 2009.

[Silva 10] Carlos Eduardo Silva. Reverse engineering of rich internet applications. Relatório técnico, Universidade do Minho, 2010.

[Systa 01] T. Systa. Dynamic reverse engineering of java software. Relatório técnico, University of Tampere, Finland, 2001.

[Thimbleby 08] Harold Thimbleby e Jeremy Gow. Applying graph theory to interaction design. páginas 501–519, 2008.

[Tip 95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, september 1995.

[Visser 03] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. 2003.

[Visser 04] Joost Visser e João Saraiva. Tutorial on strategic programming across programming paradigms. Em *8th Brazilian Symposium on Programming Languages*, Niteroi, Brazil, May 2004.

[Walrath 04] Kathy Walrath, Mary Campione, Alison Huml, e Sharon Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Prentice-Hall, 2nd edição, 2004.